Université de Technologie de
Belfort-Montbéliard


Intelligence artificielle : concepts
fondamentaux et langages dédiés

IA41

# Compte-rendu de projet

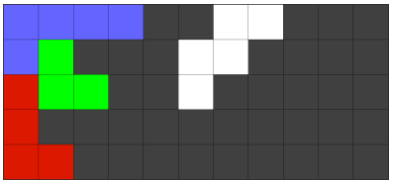
Quentin Balezeau
Estouan Gachelin
Guillaume Py
Kevin Le Saux

# Table des matières

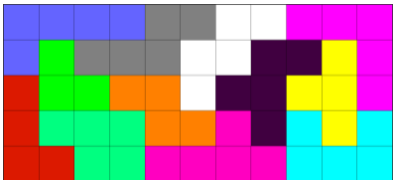
# 1. Rappel de l’énoncé

Le sujet traité est le jeu IQ Puzzler, dont l'objectif est de couvrir intégralement le plateau de jeu à l'aide de toutes les pièces disponibles. Étant donné le nombre considérable de combinaisons possibles, le jeu est structuré en différents « niveaux » de difficulté. Ces niveaux sont définis par des configurations initiales où certaines pièces sont déjà positionnées sur le plateau, et le défi consiste à compléter l'agencement sans déplacer ces pièces préétablies. Les niveaux varient de facile à difficile. La finalité de cet exercice est de développer un algorithme capable de résoudre les énigmes proposées par le IQ Puzzler. Ce rapport détaillera notre approche pour surmonter ce défi, en décrivant les obstacles rencontrés et les stratégies mises en œuvre pour les résoudre.

Plateau vide



Exemple de niveau



Exemple de solution du niveau

Le nombre de combinaison possible dans  le jeu IQ Puzzler se compte en millier, il prendrait beaucoup de temps de toutes les tester même si les problèmes permette de réduire le nombre de solution possible. Il est donc nécessaire de filtrer les possibilités et de savoir quand ne pas les tester pour éviter les calculs long et inutile.

# 2. Formalisation du problème

1. Objectif du jeu : Placer toutes les pièces du puzzle dans un espace défini (comme une grille ou un cadre) sans chevauchement et sans laisser d'espace vide.

2. Définition de l'espace de jeu : Spécifier les dimensions de l'espace dans lequel les pièces doivent être placées. Par exemple, une grille de 10x10 cases.

3. Description des pièces : Chaque pièce du puzzle peut être décrite en termes de sa forme, de sa taille, et de son orientation possible. Par exemple, une pièce peut être en forme de "T", couvrir 5 cases, et être pivotée dans quatre orientations différentes.

4. Règles de placement : Établir des règles pour le placement des pièces, comme :
   - Les pièces ne doivent pas se chevaucher.
   - Les pièces doivent être entièrement contenues dans l'espace de jeu.
   - Les pièces peuvent être tournées mais pas reflétée.

5. Variables et paramètres :
   - Position de chaque pièce (coordonnées x, y dans la grille).
   - Orientation de chaque pièce (par exemple, 0°, 90°, 180°, 270°).
6. Hypothèses :
   - Toutes les pièces peuvent être placées dans l'espace de jeu sans laisser d'espace vide.
   - Il y a au moins une solution au puzzle.
7. Stratégie de résolution :
   - Approche par essai et erreur (essayer toutes les positions possibles et s'arrêter quand solution trouver, si pas trouvée = pas de solution).
   - Heuristiques, comme placer les plus grandes pièces en premier.

Validation : Tester la solution pour s'assurer que toutes les pièces sont placées conformément aux règles et que l'espace de jeu est entièrement rempli.

# 3. Analyse du problème,

Le but principal de l'analyse du problème, comme dans l'exemple du jeu IQ Puzzler, est de mieux comprendre et structurer le problème pour trouver une solution plus efficace et systématique. Voici les objectifs spécifiques de cette analyse :

1. Clarifier le Problème : Identifier précisément ce qui doit être résolu. Cela permet de se concentrer sur les vraies difficultés plutôt que sur les symptômes ou les aspects périphériques.

2. Comprendre les Causes : Examiner les raisons pour lesquelles le problème existe. Cela aide à cibler les efforts de résolution sur les causes profondes plutôt que sur les effets.

3. Définir les Objectifs de la Solution : Établir clairement ce que l'on cherche à atteindre. Dans le cas d'IQ Puzzler, l'objectif est de placer toutes les pièces correctement le plus rapidement possible.

4. Identifier les Contraintes et les Ressources : Prendre en compte les limitations (comme le temps ou les règles du jeu) et les ressources disponibles, ce qui est crucial pour élaborer une stratégie réalisable.

5. Développer des Stratégies de Résolution : Proposer des méthodes ou des approches pour résoudre le problème. L'analyse aide à concevoir des stratégies plus ciblées et efficaces.

6. Optimiser le Processus de Résolution : Réduire le temps et l'effort nécessaires pour résoudre le problème. Par exemple, en éliminant les solutions impossibles dès le début dans le cas d'IQ Puzzler.

7. Apprentissage et Amélioration : Permettre l'extraction de leçons et de meilleures pratiques qui peuvent être appliquées à des problèmes similaires à l'avenir.

En résumé, l'analyse du problème est un outil essentiel pour aborder de manière structurée et efficace la résolution de problèmes complexes, en aidant à comprendre en profondeur le problème, à définir des objectifs clairs, et à élaborer des stratégies adaptées.

Analyse du problème pour le jeu IQ Puzzler avec la stratégie choisie de tester les solutions possibles jusqu'à trouver une réponse, tout en éliminant les options impossibles pour gagner du temps

1. Identification du problème :
    - Problème Principal: Trouver la bonne disposition des pièces dans le jeu IQ Puzzler pour remplir complètement l'espace de jeu sans chevauchement ni espace vide.
    - Difficulté: Le nombre de combinaisons possibles rend la recherche d'une solution par tâtonnement très chronophage.

2. Collecte d'informations :
    - Comprendre les règles du jeu, la forme des pièces, et les contraintes de placement.
    - Évaluer le nombre total de configurations possibles en fonction du nombre de pièces et de leurs orientations possibles.

3. Analyse des causes du problème :
    - Le problème réside dans le nombre élevé de combinaisons possibles, ce qui rend la méthode d'essai et erreur inefficace sans une stratégie de réduction des options.

4. Définition des objectifs de la stratégie de résolution :

- Trouver une méthode systématique pour tester les configurations tout en réduisant le nombre de combinaisons à examiner.
- Éliminer rapidement les configurations impossibles pour gagner du temps.

5. Élaboration de la stratégie de résolution :
    - Commencer par placer les pièces les plus grandes ou les plus complexes, car elles ont moins de positions et d'orientations possibles.
    - Utiliser une approche de backtracking : si une pièce ne s'adapte pas, revenir en arrière et essayer une autre pièce ou orientation précédente.
    - Identifier et éliminer les configurations qui violent clairement les règles (comme le chevauchement des pièces).

6. Identification des ressources nécessaires :
    - -Temps et patience pour tester différentes configurations.
    - Éventuellement, un système pour enregistrer et suivre les configurations déjà testées.

7. Mise en œuvre et test :
    - Appliquer la stratégie étape par étape sur le puzzle.
    - Noter les progrès et ajuster la stratégie au besoin.

8. Évaluation et ajustement :
    - Analyser les résultats de la stratégie mise en œuvre.
    - Ajuster la méthode en fonction des succès et des obstacles rencontrés.

Cette analyse du problème et de la stratégie de résolution pour le jeu IQ Puzzler met en évidence une approche structurée pour aborder un problème complexe, en se concentrant sur l'élimination des options impossibles et sur l'optimisation du processus de recherche de solutions.

# 4. Méthode proposée

La méthode utilisée est une fonction qui va essayer toutes les possibilités existantes. Pour cela, nous plaçons une pièce en partant du haut à gauche, puis nous utilisons ce nouveau tableau et recommençons le placement d'une pièce. Si nous nous retrouvons dans une situation où plus aucune pièce ne peut être placée, on enlève la dernière pièce placée et on en place une nouvelle à la place.

Le programme peut donc être séparé en plusieurs parties. Tout d'abord, il est important de savoir quelle est la position où nous voulons placer la pièce. Pour cela, nous partons de la position de la dernière pièce placée et nous cherchons la prochaine case vide, en parcourant les tableaux de haut en bas et en se décalant vers la gauche à chaque ligne.

```python
## @brief This function advances to the next empty cell
# @param table The game board
# @param position The current position
# @return The position of the next empty cell
def avancer_case_vide(table, position):
    i, j = position
    while i < len(table):
        if table[i][j] != 0:
            next_position = (i + 1, j)
            if next_position[0] == len(table):
                next_position = (0, j + 1)
            i, j = next_position
        else:
            break
    return i, j
```

Ensuite, nous testons de placer une pièce. Nous avons alors une vérification qui a lieu : nous vérifions si la pièce ne sort pas du terrain et si la pièce n'essaie pas d'être placée sur une pièce déjà existante. Si la pièce a réussi à être placée, nous appelons récursivement la fonction avec le nouveau tableau. Si, au début d'une récursion, nous avons placé l'intégralité des pièces, nous avons alors une solution.

```python
## @brief This function finds a solution to the game using brute force
# @param affichage The game interface
# @param used_pieces The list of used pieces
# @param table The game board
# @param position The current position
# @return True if a solution is found, False otherwise
def brutforcefct(affichage: interface.Interface, used_pieces, table, position=(0, 0)):
    if 0 not in used_pieces:
        print("Solution trouvée:")
        plateau_solution = jeu.Board(len(table), len(table[0]))
        plateau_solution.board = table
        plateau_solution.printBoard()
        return True
    if not verife_case_isolee(table):
        return False
    i, j = avancer_case_vide(table, position)
    temp_table = jeu.Board(len(table), len(table[0]))
    temp_table.board = [row[:] for row in table]
    for piece_id in range(1, 13):
        if used_pieces[piece_id - 1] == 0:
            current_piece = jeu.Piece(piece_id)
            for _ in range(2):
                for _ in range(current_piece.rotation):
                    position = i, j
                    m = 0
                    while current_piece[m][0] == 0:
                        m = m + 1
                    position = position[0] - m, position[1]
                    if table.canPlaceShape(current_piece, position):
                        temp_table.placeShape(current_piece, position)
                        next_position = (i + 1, j)
                        if next_position[0] == len(table):
                            next_position = (0, j + 1)
                        updated_used_pieces = used_pieces[:]
```

```
                    updated_used_pieces[piece_id - 1] = 1
                    affichage.board.board = temp_table.board
                    t = threading.Thread(target=brutforcefct,
                                         args=(affichage, updated_used_pieces, temp_table, next_position))
                    t.start()
                    t.join()
                    affichage.remove_shape(piece_id)
                    used_pieces[piece_id - 1] = 0
                    temp_table.board = [row[:] for row in table]
                current_piece.turnClockwise()
            if current_piece.can_miror:
                current_piece.mirror()
            else:
                break
```

Afin de limiter les itérations inutiles, nous avons tout de même une vérification qui regarde si la dernière pièce placée ne crée pas une case vide isolée, entourée par des cases remplies, ce qui serait alors une situation où il n'existe pas de solution.

```
## @brief This function checks if a cell is isolated
# @param plateau The game board
# @param ligne The row of the cell
# @param colonne The column of the cell
# @return True if the cell is isolated, False otherwise
def case_isolee(plateau, ligne, colonne):
    voisins = [(ligne - 1, colonne), (ligne + 1, colonne), (ligne, colonne - 1), (ligne, colonne + 1)]
    for voisin_ligne, voisin_colonne in voisins:
        if len(plateau) > voisin_ligne >= 0 and 0 <= voisin_colonne < len(plateau[0]):
            if plateau[voisin_ligne][voisin_colonne] == 0:
                return False
    return True

## @brief This function checks if there is an isolated cell in the board
# @param table The game board
# @return True if there is no isolated cell, False otherwise
def verife_case_isolee(table):
    for ligne in range(len(table)):
        for colone in range(len(table[0])):
            if table[ligne][colone] == 0:
                if case_isolee(table, ligne, colone):
                    return False
    return True
```

Fonctions de vérification

# 5. Jeu d’essai du programme

Nous avons tester notre programme dans plusieurs situations pour vérifier s’il est capable de résoudre des problèmes donnés et s’il les résout assez rapidement pour que cela soit exploitable. Nous allons voir quelque cas concret et voir comment l’algorithme se débrouille.

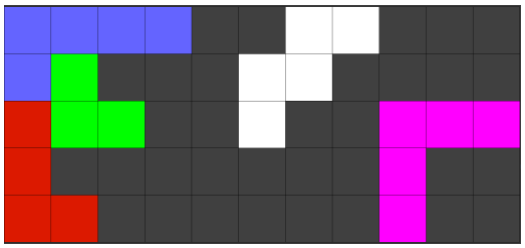
Pour chaque essaie, comme dans le jeu initiale, nous donnons une situation donné, un problème à résoudre. Puis nous laissons le programme trouver des solutions. Le programme peux trouver plusieurs solutions différentes, elles seront écrites dans le terminal.

Essai n°1 :

Dans cette situation nous essayons de voir si le programme comprend qu'il n'y a aucune solution possible. Grace à certaine vérification le programme sait qu'il n'y a aucune solution lorsque que certaine case sont inaccessible cette à dire qu'aucune pièce ne peux ce placer dans ces cases.

Dans ce cas de figure nous avons placé une pièce de façon a bloquer le coin inferieur gauche, ce qui rend impossible la résolution du problème. Le programme reconnais vite qu'il n'y a aucune solution et arrête donc la recherche de solution.

Situation initiale :



Pas de solution possible, la pièce violette bloque toute solution possible car il n'existe pas de pièce en forme de carré donc il n'y a pas de solution trouvable, dans ce cas de figure le programme arrête la recherche dès qu'il voit qu'il n'y a pas de solutions possible.  Vu le cas de figure le programme ne trouve logiquement aucune solution.

Lorsqu'une telle impasse est identifiée, l'algorithme de backtracking intervient : le programme remonte d'un niveau, c'est-à-dire qu'il retire la dernière pièce placée et essaie une autre configuration. Cela permet de ne pas perdre de temps avec des chemins qui ne mènent à aucune solution.

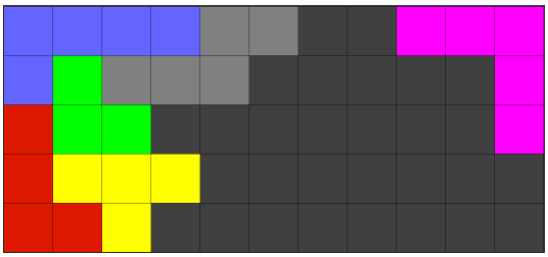
La stratégie de backtracking repose sur l'hypothèse qu'une fois qu'un blocage est atteint, il n'y a pas de raison de continuer plus loin dans cette direction. Cela est particulièrement vrai pour les puzzles spatiaux comme l'IQ Puzzler, où la forme des pièces et l'espace limité imposent des contraintes strictes sur les configurations possibles.

Les arbres de solutions, ou plutôt les branches de ces arbres, qui mènent à des impasses sont

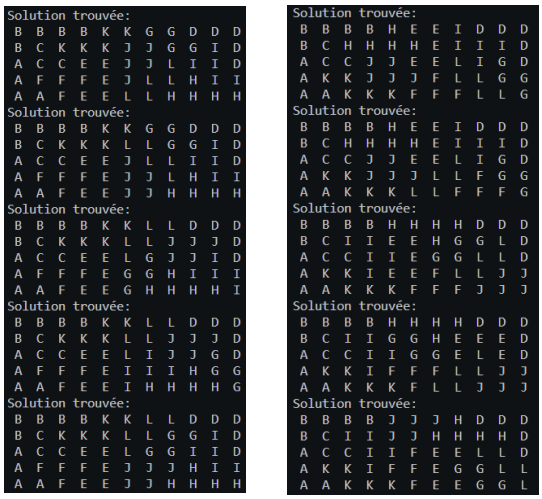
abandonnées, et l'algorithme revient à un point de décision antérieur pour essayer une alternative différente. Ce processus continue jusqu'à ce que toutes les solutions possibles soient trouvées ou que toutes les combinaisons aient été épuisées.

Essai n°2 :

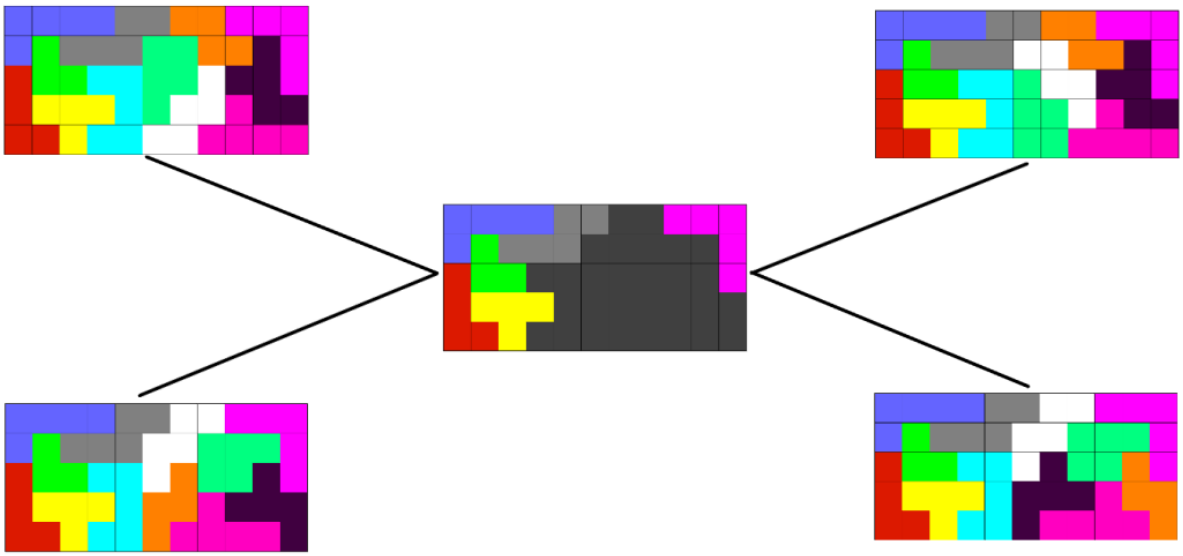
Situation initiale :



Plusieurs solution possible ont été trouver en quelques secondes. Le programme ne s'arrête pas à la première solutions trouver, il s'arrête une fois que toutes les solutions sont trouvées. Certain arbre de solutions sont impossible à finir. Exemple : Si une pièce bloque les solutions, un peu comme dans le cas précédent en bloquant certaines cases en rendant impossible de positionné les autres pièces. Dans ce cas de figure on revient là où l'on a posé la pièce problématique. Voici plusieurs solutions possible au cas de figure ci-dessus :




Les différentes solutions ne sont pas exhaustive, mais l'ont peu remarquer quelles peuvent venir d'une même base. Par exemple dans ce cas de figure la position de la pièce F amène à trouver plusieurs solutions. Le programme va donc explorer toutes les solutions possible a partir d'une branche, ici appelons la « branche F » on peut voir quand continuons dans cette branche, plusieurs solutions sont trouvables.

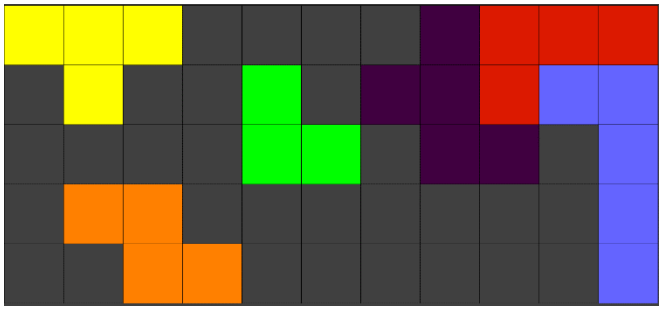


On peut donc se rendre compte que plus il y a de pièces placées à la base du problème plus le problème est simple. La difficulté d'un problème vient donc du nombre de possibilité de solution qu'il offre, plus il y a de pièce à placé plus il est difficile à résoudre.

Essai n°3 :

Dans les cas où il n'y a pas de solution mais qu'il y a pas de cases « bloqué » par d'autre pièce, le programme testera le peu de solution possible assez rapidement et arrêtera la recherche car aucune solution n'est trouvable. Dans les cas de figure ou les problèmes n'ont pas de résolutions possibles il y a généralement beaucoup de pièces placées pour empêcher les autres de ce positions comme il faut. Il est donc plutôt simple pour le programme de ce rendre compte qu'il n'y a pas beaucoup de solution du au fait qu'il y ai un nombre limité de pièces à placées.

Situation initiale :



Aucune solution n'a été trouvées il est donc simplement impossible de résoudre ce problème.

# 6. Difficulté rencontré

Problème sur l'interface et l'optimisation au début du programme, l'interface était simplement trop instable et très capricieuse. Normalement ces problèmes sont réglés

# 7. Amélioration possibles

Nous aurions pu peut être encore accéléré la calcule de notre programme, en parallélisant le calcul de certaine branche de solution. Pour optimiser davantage le calcul et l'exécution de notre programme conçu pour résoudre le IQ Puzzler, il serait judicieux de mettre en œuvre des techniques de parallélisation. Cette approche consisterait à diviser l'espace de recherche en plusieurs branches de solutions indépendantes, qui pourraient être traitées simultanément sur différents cœurs ou processeurs. En distribuant ainsi le processus de recherche brute, nous réduirions significativement le temps nécessaire pour arriver à une solution.

En outre, nous pourrions exploiter le multithreading pour permettre à notre fonction de force brute de s'exécuter sur plusieurs threads. Chaque thread pourrait prendre en charge une partie distincte de l'espace de recherche, permettant une exploration concurrente des différentes configurations possibles. Cette stratégie est particulièrement efficace sur des systèmes multicœurs, où les threads peuvent réellement s'exécuter en parallèle, menant à une réduction du temps d'exécution global.

Cependant, il convient de noter que le multithreading et la parallélisation requièrent une conception soignée pour éviter les conditions de concurrence et les problèmes de synchronisation. Les ressources partagées, telles que la mémoire, doivent être gérées avec précaution pour prévenir les conflits d'accès. De même, il serait nécessaire de définir des points de synchronisation pour que les threads ou les processus parallèles puissent combiner leurs résultats de manière cohérente.

Pour gérer ces défis, nous pourrions implémenter des verrous, des sémaphores ou d'autres mécanismes de synchronisation, ainsi que des techniques de décomposition de tâches qui minimisent la dépendance entre les threads. Avec une telle infrastructure en place, notre programme ne serait pas seulement capable de résoudre les puzzles de manière plus efficace, mais il serait également bien adapté pour évoluer sur des systèmes informatiques de haute performance, exploitant pleinement le potentiel des architectures modernes multicœurs.

Une autres voie d'amélioration possible aurait été l'optimisation de notre programme, aller à l'essentiel, c'est déjà le cas mais notre programme en utilisant des techniques heuristiques. L'heuristique est une technique qui permet de guider la recherche de solutions de manière intelligente afin d'optimiser les performances d'un programme. En appliquant des

heuristiques dans notre programme de résolution du IQ Puzzler, nous pouvons réduire significativement l'espace de recherche et le temps de calcul nécessaire pour trouver une solution. Voici comment cela pourrait être mis en œuvre :

Évaluation de Position : Une fonction d'évaluation heuristique pourrait être conçue pour attribuer un score à chaque configuration du plateau en fonction de sa proximité avec une solution complète. Par exemple, des points pourraient être attribués pour chaque ligne, colonne ou région qui se rapproche d'un état complet sans espaces vides.

Priorisation des Mouvements : Les mouvements ou placements des pièces qui mènent à des configurations avec des scores heuristiques plus élevés seraient prioritaires. Cela oriente le programme vers les chemins les plus prometteurs et éloigne des impasses.

En conclusion, l'application d'heuristiques dans un algorithme de résolution de puzzle comme le IQ Puzzler peut non seulement accélérer la recherche de solutions, mais aussi rendre l'exploration de l'espace de solutions plus efficace et plus ciblée. Cela rend l'outil plus pratique et plus puissant pour les utilisateurs, en fournissant des solutions de manière plus rapide et plus fiable.

## 8. Perspective d’ouverture possible

Peut être en s’y prenant plus tôt on aurait surement pu mettre en place un réseau de neurone convolutif. Le plus gros problème de cette technique au-delà de la complexité de la mise en place d’un réseau similaire, le temps n’aurais pas été suffisant, le temps de le mettre en place et de le faire fonctionné nous aurais pris plusieurs semaines. Mais le plus embêtant dans tout cela aurait été l’entrainement du réseau de neurone, n’ayant pas de supercalculateur pour paralléliser les parties pour entrainer notre réseau il nous aurait fallu plusieurs mois pour que l’IA soit entrainée sous peine que notre fonction de récompense soit bien programmée et que notre réseau de neurone soit partie assez vite dans la bonne direction lors de son entrainement.

Mais pour y procéder voici comment on aurait pu y parvenir. L’IA n’a pas été implémenté par manque de performance de sa part, il était donc plus cohérant de la mettre de côté et d’optimiser ce qui fonctionnait déjà. Peut-être qu’on l’entrainera plus tard qui sait…

Étape 1 : Collecte et préparation des données

Nous commencerions par compiler un vaste ensemble de données de différents niveaux de l'IQ Puzzler, y compris les positions initiales et les solutions finales. Chaque configuration de puzzle serait numérisée en une matrice représentant le plateau de jeu, où les emplacements des pièces seraient encodés par des vecteurs caractéristiques.

Étape 2 : Conception du réseau de neurones

Un réseau de neurones convolutifs (CNN) serait approprié pour ce type de problème spatial. Les CNN sont excellents pour reconnaître des patterns visuels et spatiaux, ce qui est essentiel dans la résolution de puzzles. Nous concevrions le réseau avec plusieurs couches cachées capables de détecter les caractéristiques et les contraintes des pièces du puzzle. (Source : Internet).

On ne savait pas exactement comment s'y prendre pour créer ce genre de réseau de neurone. On a donc regarder sur internet ce qui avait déjà été fait pour essayer de comprendre et de refaire ce qui avait déjà fonctionnée.

Étape 3 : Entraînement du modèle

Nous entraînerions le réseau de neurones en utilisant l'ensemble de données préparé, en ajustant les poids du réseau via la rétropropagation. Durant cette phase, le réseau apprendrait à identifier les emplacements valides pour les pièces en se basant sur les positions initiales et les solutions connues.

En s'entrainant pendant longtemps même en plaçant des pièces aléatoirement le réseau allait finir par avoir des bonnes solutions et dans ces cas de figure (même s'il était atteint par pur hasard) il fallait bien récompenser le réseau pour qu'il comprenne que cette situation final était bonne.

Étape 4 : Amélioration et validation du modèle

Le modèle serait ensuite testé sur un ensemble de validation varier (problème de IQ Puzzler en somme) pour évaluer sa performance. Des ajustements seraient apportés en fonction des résultats, avec des techniques comme l'augmentation de données ou le réglage fin des hyperparamètres. Même si dans les fait nous allions surement le laisser s'entrainer plus longtemps pour voir s'il était capable de s'amélioré et si sur le long terme il était capable de résoudre plus simplement les niveaux dit « facile »

Défis et Solutions :

Le principal défi serait d'assurer que le réseau de neurones puisse généraliser des solutions à des puzzles jamais vus auparavant. Pour cela, nous pourrions utiliser des techniques d'apprentissage en profondeur et de renforcement pour permettre à l'IA d'explorer et

d'apprendre de nouvelles stratégies de résolution. De plus, des mécanismes de régularisation pourraient être mis en œuvre pour éviter le surajustement sur les données d'entraînement.

En conclusion, la création d'un réseau de neurones pour résoudre le jeu IQ Puzzler serait un projet ambitieux, mais réalisable. Il combinerait des techniques avancées d'apprentissage automatique avec une ingénierie logicielle réfléchie pour produire une IA capable non seulement de résoudre des puzzles complexes, mais également de fournir des insights sur les stratégies de résolution de problèmes spatiaux.

# 9. Conclusion

En conclusion, ce rapport a exploré la faisabilité et l'implémentation d'un programme intelligent pour résoudre le jeu IQ Puzzler. À travers une démarche méthodique, nous avons abordé la complexité inhérente à ce casse-tête et mis en œuvre des stratégies de résolution efficaces. L'utilisation de techniques de force brute, appuyée par des vérifications et optimisations heuristiques, a permis de réduire de manière significative l'espace de recherche et le temps de calcul. Malgré les défis rencontrés, notamment en ce qui concerne l'interface utilisateur et l'optimisation des calculs, le programme a démontré sa capacité à trouver des solutions dans divers scénarios de test.

Ce travail a donc non seulement abouti à un outil fonctionnel pour résoudre le IQ Puzzler mais a également posé les bases pour des avancées futures dans le domaine de l'intelligence artificielle appliquée aux jeux de réflexion. En continuant sur cette voie, il est envisageable que des programmes encore plus performants et intuitifs voient le jour, bénéficiant aux amateurs de puzzles ainsi qu'aux chercheurs en IA.